Implementation of a Python library that generalizes the CLUE algorithm

Presented by: Simone Balducci

Alma Mater Studiorum-Università di Bologna

November 1st 2022

Goals of the project

The goals of this project were to:

- Generalize CLUE to N dimensions
- Bind the algorithm to Python
- Write a Python class "clusterer" that executes CLUE
- Deploy the algorithm to a PyPi library

Generalization to N dimensions

To generalize the code, several steps were needed:

- add the number of dimensions, Ndim, as a template parameter for all the classes
- introduce a new metric $\longrightarrow$ N-dimensional euclidian metric
- remove the parameters specific to the detector's geometry
- redefine the tiles
- rewrite the algorithms for the local density and the distance to higher
- rewrite the mainRun functions

Tiles

- originally, the tiles where calculated based on the detector's geometry
  - ➤ now we want them to be calculated based on the points
- the number of tiles is calculated based on the desired average number of points per tile $\longrightarrow$ new constructor's input parameter
- the size of the tiles in each dimensions is calculated depending on the coordinate's range

$$NTiles = 10,\ x_{min} = 1,\ x_{max} = 4 \longrightarrow \boxed{xbin_{size} = 0.3}$$

- the resulting tiles are all identical parallelepipeds

Local density and distance to higher

- the original algorithm iterated through x and y bins
- we must find a way to iterate through a generic number of dimensions
- we define a recursive function
  - it calculates all the N dimensional bins in the given range
  - executes the calculation (local density or NH) in each one

```
template <uint8_t N_>
void for_recursion(std::vector<int> &base_vector,  std::vector<int> &dim_min, std::vector<int> &dim_max,
    if constexpr (N_ == 0) {
        int binId = lt_.getGlobalBinByBin(base_vector);
        // get the size of this bin
        int binSize = lt_[binId].size();

        // iterate inside this bin
        for (int binIter = 0; binIter < binSize; ++binIter) {
            int j = lt_[binId][binIter];
          // query N_{dc_}(i)
          float dist_ij = distance(point_id, j);

          if(dist_ij <= dc_) {
             // sum weights within N_{dc_}(i)
             points_.rho[point_id] += (point_id == j ? 1.f : 0.5f) * points_.weight[j];
          }
        } // end of interate inside this bin
        return;
     } else {
         for(int i = dim_min[dim_min.size() - N_]; i <= dim_max[dim_max.size() - N_]; ++i) {
              base_vector[base_vector.size() - N_] = i;
              for_recursion<N_-1>(base_vector, dim_min, dim_max, lt_, point_id);
         }
     }
  }
```

Local density and distance to higher

- we then use the recursive function to calculate the local density (or NH)

```
void calculateLocalDensity(tiles<Ndim>& tiles) {
  // loop over all points
  for(int i = 0; i < points_.n; ++i) {
    // get search box
    std::array<std::vector<float>,Ndim> minMax;
    for(int j = 0; j != Ndim; ++j) {
      std::vector<float> partial_minMax{points_.coordinates_[j][i]-dc_,points_.coordinates_[j][i]+dc_};
      minMax[j] = partial_minMax;
    }
    std::array<int,2*Ndim> search_box = tiles.searchBox(minMax);

    // loop over bins in the search box(binIter_f - binIter_i)
    std::vector<int> binVec(Ndim);
    std::vector<int> dimMin;
    std::vector<int> dimMax;
    for(int j = 0; j != (int)(search_box.size()); ++j) {
      if(j%2 == 0) {
        dimMin.push_back(search_box[j]);
      } else {
        dimMax.push_back(search_box[j]);
      }
    }

      for_recursion<Ndim>(binVec,dimMin,dimMax,tiles,i);
  } // end of loop over points
}
```

mainRun and run functions

- in C++ template parameters must be known at compile time
- in order to bind mainRun, the number of dimensions can't be a variable
- we define many "run" functions, for which Ndim is known and constant

```
////////////////////
//////  Run.h  //////
////////////////////
std::vector<std::vector<int>> run1(float dc, float rhoc, float outlier, int pPBin,
        std::vector<std::vector<float>> const& coordinates, std::vector<float> const& weight) {
    ClusteringAlgo<1> algo(dc,rhoc,outlier,pPBin);
    algo.setPoints(coordinates[0].size(), coordinates, weight);

    return algo.makeClusters();
}

std::vector<std::vector<int>> run2(float dc, float rhoc, float outlier, int pPBin,
        std::vector<std::vector<float>> const& coordinates, std::vector<float> const& weight) {
    ClusteringAlgo<2> algo(dc,rhoc,outlier,pPBin);
    algo.setPoints(coordinates[0].size(), coordinates, weight);

    return algo.makeClusters();
}

std::vector<std::vector<int>> run3(float dc, float rhoc, float outlier, int pPBin,
        std::vector<std::vector<float>> const& coordinates, std::vector<float> const& weight) {
    ClusteringAlgo<3> algo(dc,rhoc,outlier,pPBin);
    algo.setPoints(coordinates[0].size(), coordinates, weight);

    return algo.makeClusters();
}
```

mainRun and run functions

- mainRun takes Ndim as input parameter and uses it to call the right run function

```
std::vector<std::vector<int>> mainRun(float dc, float rhoc, float outlier, int pPBin,
            std::vector<std::vector<float>> const& coords, std::vector<float> const& weight, int Ndim) {
    // Running the clustering algorithm //
    if (Ndim == 1) {
        return run1(dc,rhoc,outlier,pPBin,coords,weight);
    }
    if (Ndim == 2) {
        return run2(dc,rhoc,outlier,pPBin,coords,weight);
    }
    if (Ndim == 3) {
        return run3(dc,rhoc,outlier,pPBin,coords,weight);
    }
```

- the compiler is happy

Binding

- the binding has been done with the use of Pybind11
- the binded function is mainRun

```
////////////////////////////////
//////  Binding module  //////
////////////////////////////////
PYBIND11_MODULE(CLUEsteringCPP, m) {
    m.doc() = "Binding for CLUE";

    m.def("mainRun", &mainRun, "mainRun");
}
```

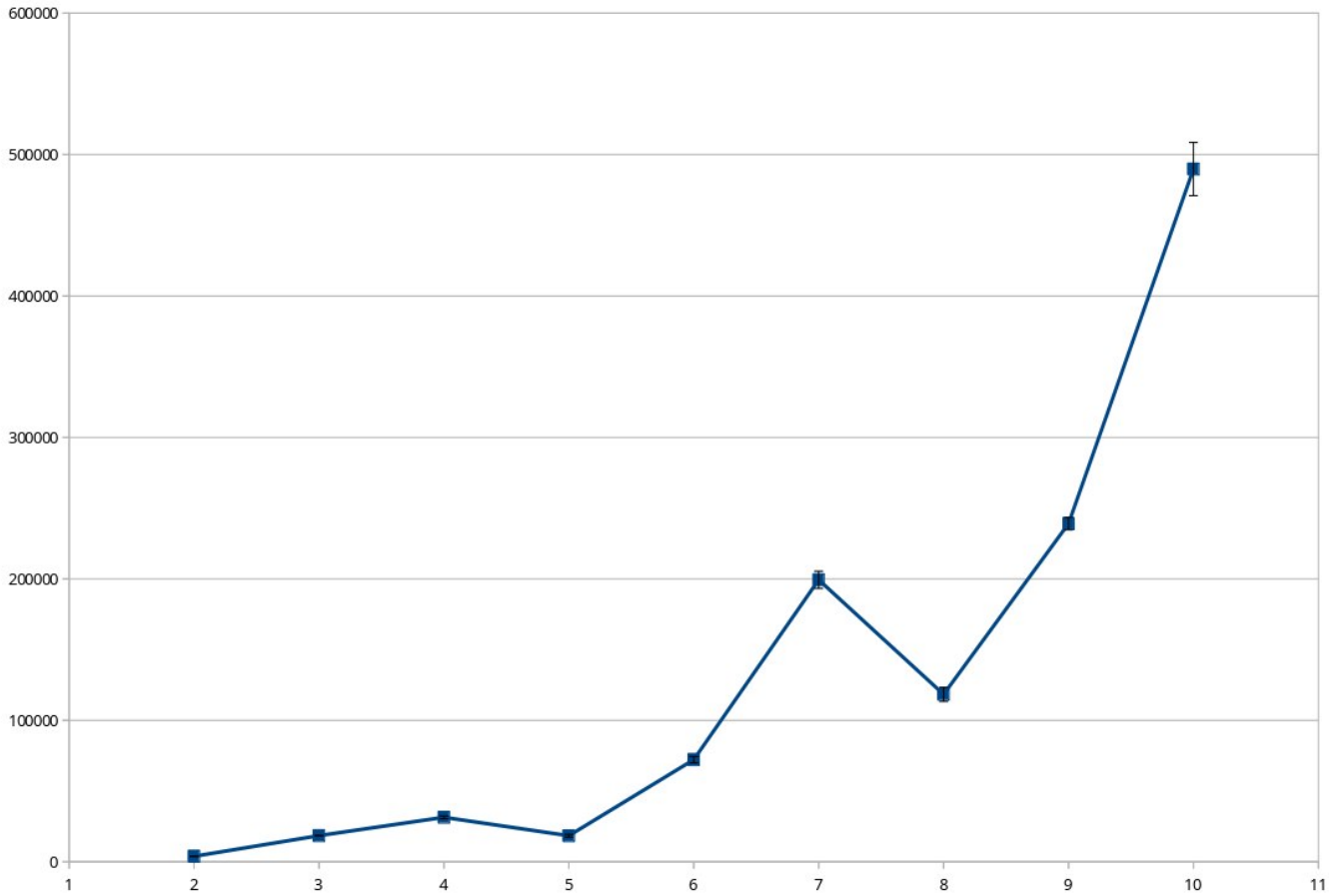
The *clusterer* class

In the library is defined a class named clusterer
The class contains some methods:

- constructor
- readData
- runCLUE
- inputPlotter
- clusterPlotter
- toCSV

Execution times for different numbers of dimensions

- the execution times have a weird trend
- nonetheless, the trend is upwards



Python library

- The library has been called CLUEstering
- It can already be found on PyPi

`https://pypi.org/project/CLUEstering/`

Possible future developements

- better documentation and docstrings
- doing a preliminary calculation of the point density
  - first estimation for the parameters
  - tiles with variable size
- optimization on the C++ side for better performance
- (maybe) run on GPU